72-(

A Guide to 15-100

R. N. Chanon

Department of Computer Science
Carnegie-Mellon University

August, 1972

Contents

Introduction: Why and How this Guide Came to Be

I wrote these pages because blank expressions bother me! By that I mean, I don't like to see a classroom filled with people who are either unwilling or unable to answer a question and who manifest their state of mind by a sort of nebulous stare - the blank expression. This unhappy situation is probably just as disconcerting to students as it is to me. The reason for it, I think, relates to the nature of the questions which I ask. They're meant to be non-trivial. I feel that class time is valuable and shouldn't be wasted by simply presenting material which can be read from a textbook. Instead, time should be spent discussing the implications and intent of the assignment. This means answering questions and solving problems. Unfortunately, existing course materials - textbooks and programming problems - don't seem to prompt much inquiry as to either the implications or the intent of an assignment. Students seem to think it's sufficient to simply read some assigned text and digest only its content. Hopefully, this little guide will help change that attitude.

In the pages which follow, I've been critical of what seems to be about the best material for an introductory course in computing. As I see it, the marketed textbooks are abysmally bad. They tend not to provoke much inquiry into what programming is about, and frequently address nothing more than the syntax of a programming language. Hence, I have tried to expose some essential ideas from amid all the verbiage.

Note, however, that this guide is NOT a textbook. It was written to specifically accompany 15-180 at CMU. Its primary purpose is to provoke questions about programming and problem solving; nothing more, nothing less.

I also emphasize the importance of asking questions.

ASK QUESTIONS!

Questions allow your instructors to talk about issues which are important to you. They can also prevent him from wasting your time while he discusses things you already understand. This guide should prompt lots of questions.

Also included are some programming problems and their analyses. Each is accompanied by a few sentences describing what motivated me to include the problem and what I expect you to learn from it. The texts of complete, running PLAGO programs accompany them all. Understand them!

There are even a few pages of motherly advice about how to allocate your time while working on the problems, along with some words about how to prepare and submit programs.

R. N. Chanon
August, 1972

Some Words about 15-100

15-100 is offered every semester to students of engineering and science at CMU. Because there are no prerequisites for 15-100, and because both freshmen and graduate students take the course, the backgrounds of the students are diverse - to say the least. Therefore, since essentially the only information your instructor has about you is your name, it is vital that you ask questions about the material which you don't understand - more about this later.

The purpose of this course is to teach you to solve problems using a digital computer. By the end of the course, you should be able to:

1. Recognize when a computer is an appropriate tool for solving a problem.
2. Define a problem precisely and formulate an explicit process for solving it.
3. Write such a process as a program in the PLAGO programming language.
4. Determine whether a program actually does the task it was intended to do.
5. When a program does not perform as expected, alter it so that it does.

The course tries to present a large number of problems and asks how a computer might be used to help solve them. Hence, problem solving and the use of a computer as a tool to help solve problems is the real thrust of the course. The details of creating syntactically correct PLAGO programs, punching or marking cards, and submitting programs are of only ancillary interest.

The course meets three times a week for one lecture and two problem solving/question answer sessions - called recitation sessions. The lectures are intended to present "general", but vital information about both problem solving and programming. They are not to be ignored. Recitation sessions will be used by your instructor to discuss PLAGO, problems, material from the lectures, and, in general, anything of interest to the course. These sessions, however, should be driven by questions. If you don't ask questions, there are very few things which an instructor can do except give quizzes, read to you from the textbook, or present more problems. If you don't ask questions, recitation sessions become a waste of your time. If you don't intend to ask any questions, you might just as well not go to class. Your presence will just add another warm body to an already over-heated classroom.

Besides class meetings, you are asked to write algorithms to solve several problems and to represent these algorithms as PLAGO programs and to run them on CMU's computer. They are important. Do them!

Finally, 15-100 requires that you take a final examination and a mid-term. See the section which discusses grades and grading to find out how these exams and the rest of your performance will be evaluated.

That's all I wish to say about 15-100!

Textbooks: Which to Buy and What They're Good For
(Besides the obvious of course!)

Buy These:

(1) A Short Introduction to the Art of Programming
by E. W. Dijkstra

(2) PL/I Programming in Technological Applications
by G. F. Groner

(3) PLAGO/360 User's Manual

(4) A Guide to 15-100
by R. N. Chanon

What they're good for:

The book by Dijkstra (referred to hereafter as EWD316) is the best introduction to programming with which I am acquainted. It addresses what seem to be the fundamental issues of the discipline in a clear, concise and careful way. The text isn't encumbered with the syntactic and semantic details of a particular programming language. He emphasizes the task of finding and developing algorithms as THE fundamental issue in programming. I think the book is excellent!

Unfortunately, as a textbook, EWD316 can be used in the wrong way. First of all, material is presented in such a coherent way that a student might gain a false sense of security about his understanding. It all looks so easy - especially in the first three sections. Don't be mislead, however. The text is somewhat like the Bible in the sense that it is easy to read but difficult to understand in terms of the real depth that is present. Even though the assignments from the book will be short, study them carefully. Don't fall into the trap of feeling "cheated" if you think you understand the text after just one reading. The chances are, you really don't!

Secondly, the book contains too few exercises. In the pages which follow, that problem will hopefully be remedied.

The book by Groner, "PL/I Programming in Technological Applications", is meant to be a source for information as to the syntax and semantics of the programming language which you will use to implement your algorithms. It contains numerous completely worked examples, as well as carefully prepared summaries of the features of the language. The examples are related to many algorithms which are commonly used in engineering and science. Many of the algorithms, however, are poorly developed. The book also contains an enormous amount of verbiage which won't be relevant to the course. Therefore, you should rely on your recitation instructor to direct your attention to those parts which are important.

Lectures

The lectures for 15-100 present information of general relevence to computing, problem solving, and the administration of the course. In the first two categories, most of the detail is omitted - rightly so - and left to the recitation sessions. In particular, the lectures will tell you how to go about solving the problems. You may not believe it, but the way you approach a programming assignment can have a tremendous effect on the amount of time you spend on it. In the last category, announcements of due date changes for the programming problems are made. The lectures are carefully planned to focus your attention on what we feel are the important issues. They are important. The lectures can also be inspirational - indeed, there are those who believe that that's all a lecture can be.

Attend them.

Recitation Sessions

Recitation sessions should be driven by questions.

Enough! Be advised.

The PLAGO manual describes the dialect of PL/I in which you will write your programs. The syntactic and semantic descriptions are clear, but the examples of complete programs which appear in the appendix are bad.

Do you understand what the phrase "syntax of PL/C" means? Are you going to ask about it?

Grades and Grading Policies

<u>You will have the following opportunities to EARN points:</u>

*Programming Problems

2 at 30 points	60
4 at 20 points	80
6 at 10 points	60
	200

*Exams

Midterm (mean about 55-60)	100
Final (mean about 110-120)	200
	300

*Recitation

Recitation performance	50

*Basic points for semester $\boxed{\text{550 points}}$

You may earn bonus points for turning the 20 and 30 point problems in early:

1 point for each two days

(upto a ceiling of twenty per-cent of the value of the problem!)

<u>You will have the following opportunities to LOSE points:</u>

*Cheating: all credit for the thing on which you were cheating

*Turning problems in late:

1 point for each two days

*Computing too much:

one point for each dollar more than the limit
used in each month

The final grade will be assigned on the basis of the following scale

475-550	A
360-474	B
250-359	C
200-249	D
000-200	R

Policy Statement on Cheating and Course Help

With regard to homework, quizzes, and exams, cheating will not be tolerated. Anyone caught cheating on a problem will receive zero credit for the problem. Anyone caught cheating on an exam will recieve zero credit for the exam. It is recognized that student B can cheat from student A without A's knowledge. In such a case, A must prove his innocence. Protect your hard work from parasites!

When you come to an exam, do NOT sit next to the people you have studied with. Your argument that your answer is just like your friend's because you study together will be much more convincing if you don't sit together during an exam.

Some students will find themselves unable to complete a problem on time or at all. Such situations allow the student three choices: first, copy someone else's project and hope he is not caught; second, give up and put the course; third, see your instructor. The second implies an R or a withdrawal, if possible. We intend that the first case will also imply an R. Hence, the student's logical choice should be the third alternative (it can't be worse). Your instructor's door is always open, and the results of a visit may prove beneficial.

Postponements of due dates are possible. If you turn in your assignments late without discussing the situation with your instructor, your grade will be decreased by an appropriate number of points (see above).

You may discuss all problems (NOT exams!) unless otherwise specified by your instructor. Student discussion is fruitful and encouraged, but all programs must be written by the individual student. That is, you may talk with anyone (including your instructor) about assigned problems, but the actual writing of the program must be done by you.

Computers and Computing

The programming language taught in this course is PLAGO (FORTRAN conversion will be available at the end of the semester for those who want it). PLAGO runs on CMU's IBM 360 model 67. Unfortunately, computer time is a scarce resource and it is not possible to provide each student with an unlimited amount of computer time. Therefore, each student in 15106 will be expected to plan his time so that he can live within two kinds of restrictions:

1) A limit on the number of programs run each day. This will be enforced by the 360: after you have used up your limit, it won't run any more of your programs.

2) A limit on the dollar value of your computer usage each month. This will be enforced by your instructor: you lose one point for every extra dollar each month. The cost of each program is printed at the end of each job, so YOU can keep track of your usage. The exact limits will be announced at the first lecture. The cost limit will be generous - most students should require only 75 per-cent of the allotment.

Note that these are upper limits and you are NOT guaranteed to be able to get this much service. You are competing with many other users for a resource that is in short supply. Indeed, there will be times (especially the day before a problem is due) when the system can't give as much service as is requested.

If you are excited about computing and want to work on extra problems of personal interest, see your instructor. We will try to make arrangements for you to use one of the less congested computers on the campus.

What is a "Solution" to a Problem?

A solution to a programming problem is a working, documented program. It must:

1) get the right answer, even on special cases and with bizarre sets of data we might construct.

2) be reasonably efficient (don't go overboard on this point!)

3) include program documentation, i.e. your working plans for the problem.

This documentation should contain:

1) About a page of understandable English prose explaining the organization of your program, what the important variables are used for, and the representation of the data (e.g. "X is a FIXED array of length 10 which contains the x coordinates of the input").

2) A list of the procedures you will use, with a short description of what each does and how they are related.

3. A flow chart or structured description (as done often in lecture) for each such procedure.

The credit for the problem will be split between the program and the documentation as follows

If the problem is worth...	The program is worth...	The documentation is worth...
30	20	10
20	15	5
10	8	2

Programming Problems - How to keep these from ruining your weekends and your health

Each semester, 15-100 students are required to write a number of programs. These assignments differ from ordinary homework problems in that they require complete, running, and correct programs as solutions. You can't turn in slipshod, partially complete programs and expect much partial credit. This semester, your programs are to be written in PLAGO - a dialect of PL/I. The programming problems are important. Much of what you will learn from 15-100 will be a direct consequence of the experiences you have as you write and debug solutions for them. Sadly enough, however, students complain about the difficulty of the problems and that they have to spend many hours finding and debugging solutions. My answer to this complaint is quite simple:

Your approach is probably wrong.

(That's not very comforting, but it's still my reply.) With very few exceptions, the analysis required to solve the problems is simple, if you are willing to analyze the problem systematically and completely. There is no need to spend vast amounts of time. If, however, you do spend lots of time solving the problems, see your recitation instructor and explain your difficulty. He might have some suggestions.

Despite rumors to the contrary, these programming assignments are intended to force you to do the following:

(1) Find or understand an algorithm which solves the problem.

(2) Represent the algorithm as a PLAGO program.

(3) Debug the program.

(4) Convince yourself that the program solves the problem.

Items (1) and (4) are the most important issues in the above process, in the course, and in essentially all of programming - and for which I can't give you algorithms. Items (2) and (3) can be handled in a fairly mechanical way and will present only minor difficulties after you've written and run a few programs.

So, it would seem that the obvious thing to do is to spend enough time to find a complete and correct algorithm so that the remaining items require only minor attention. An hour or two of thought about the problem BEFORE writing any PLAGO statements will probably save you several hours of the total time spent finding a solution. Do this and your tenure as a student of 15-100 will only be a minor hassle - who knows, you might even like it!

How to Attack a Programming Assignment

Imagine that you have been assigned a problem - not a keypunching exercise, but a real programming problem. How can the problem be solved? Whole books have been written to help answer this question. One of the best is the small volume by George Polya entitled "How to Solve It". I recommend it as a general aid to analyzing the programming problems. More specifically, I can offer several suggestions and refer you to the programs in a later part of this guide. guide.

Things to do:

(1) Make sure that you understand what the problem asks.

Usually, the problems are posed fairly well. Hence, understanding what a problem asks isn't difficult. However, be certain that you really understand the problem statement before proceeding to the next step.

(2) Find and understand an algorithm which solves the problem.

This is the most important part of the whole process! It involves, among other things, finding an appropriate data structure and control structure for the problem.

(3) Cast your algorithm in a step-by-step way using the ideas of structured programming.

This tends to clarify your ideas and will frequently point out difficulties with your original algorithm. Never feel too proud to write a flowchart or a sequence of structured statements. The stepwise refinement technique due to Dijkstra and Wirth is particularly appropriate to this step.

(4) Write a PLAGO program which is equivalent to your flowchart or structured statements.

This step can be performed in a fairly mechanical way - it's easy. It is sometimes helpful to write several drafts of the program. Embellish your code with lots of informative comments. These comments are exceedingly useful! Comments help you to understand the mess you've created if you contract mono-nucleosis and must put the program aside for awhile. Your final draft should be complete (including system control cards). This really means that if you are lucky enough to have a girl friend who is willing to punch your cards for you, she should never have to ask you what characters to punch.

(5) Go to the third floor of Science Hall and punch or mark your cards.

This is another easy step. Examine your cards carefully before you submit them to make sure that they exactly represent your final draft. This quick check can sometimes save you several submittals.

(6) Run your program.

If it doesn't run correctly, correct it and run it again. Don't, however, just change the program "randomly". Think about what went wrong and how changes will affect the program. Repeat this process until you are convinced that your program behaves as it should (see the comments below). Make sure that you have considered all the special cases and not just the ones which our data gives you!

One final important point: TRY TO START WORKING ON A PROBLEM AS SOON AFTER IT IS ASSIGNED AS POSSIBLE, AND DON'T BE AFRAID TO WORK ON TWO PROBLEMS AT ONCE! ! ! ! ! ! ! There are almost always two problems pending at the same time.

Things not to do:

(1) Don't try writing a PLAGO program from scratch. It's almost certain to be wrong. Do so at your own risk. It has been my experience that regardless of the size or complexity of the problem, a set of structured statements or flowcharts is helpful. Should you decide to ignore this warning, expect the following things to happen:

(a) Your program will contain more syntactic and logical errors than the corresponding result had you followed the steps above.

(b) You can expect to make many changes in the program before it finally runs correctly - if it ever runs correctly.

(c) You can expect to spend lots of time at the computation center submitting programs and waiting for output. The computation center is very dull, and, frankly, isn't a very pleasant place to be.

(d) Your program will be difficult to understand, not only by someone else, but also by you.

(e) Your program will tend to be longer than the corresponding program produced by the steps above. It will also tend to cost more to run.

(f) Your understanding of programming and problem solving will tend to be weaker than had you followed the above steps - hence your grades will tend to be lower than they could have been.

That's all I have to say about this matter. Be warned.

(2) Don't spend lots of time correcting and re-correcting a program that doesn't work. The point of diminishing returns can approach quite rapidly and you can easily waste time in an unfruitful pursuit. Time is best spent making sure that your algorithms are correct!

(3) Don't wait until the day before a problem is due to start solving it. You are almost certain not to have a solution in time. Programming assignments are not like ordinary homework exercises. Not only must you solve the problem, but you must also compete for a valuable resource - computing time - to demonstrate that your program is right. Instead, start step (1) on the day the problem is announced, and finish it as soon after that as possible.

(4) Don't try to run programs the day before a problem is due. The user area is mobbed by people who have neglected the problem. It is almost impossible to get anything done under these circumstances.

That's all the motherly advice I wish to give about programming problems. Write below the phrases from the above passages which you don't understand, and ask about them.

Array Walking

In addition to the simple variable, data may also be stored into objects known as ARRAYs. An array is nothing more than a name - just like one for a simple variable - which identifies a whole collection of simple variables. Names become associated with array data structures by declaring them as such. Thus in PLAGO

```
        DECLARE A(0:100) FIXED;
```

declares A to be an array of elements A(0), A(1), ... , A(100) . Arrays are particularly useful because the symbols which name array elements can frequently name more than just one element. Thus

```
A(I)
```

names an element of A, designated by the value of I. Thus if the 101 elements of A contain numeric values, the following fragment computes, respectively, the minimum and maximum values contained within A.

```
/* compute the maximum value in A and store it into MAXA.
   compute the minimum value in A and store it into MINA */

/* set MINA and MAXA to the value of an element of A */

     MINA, MAXA = A(0) ;

     DO I = 1 to 100;
     IF A(I) < MINA THEN MINA = A(I) ;
        ELSE IF A(I) > MAXA THEN MAXA = A(I);
     END ;
```

This fragment can be made more flexible by noting that the upper bound of the DO statement can be replaced by a variable, say N. This means that if 0 <= N <= 100 then only the first N elements of A will be examined for the maximum and the minimum. Obviously, the lower bound, 0, can be made a variable as well (say M).

Now, suppose that we wish to rearrange the contents of A(0),..., A(N) such that these elements of A are in ascending order. There are many ways of doing this. One way which every beginning programmer learns is a method called the SHUTTLE SORT. It can be developed by noting that the smallest element of A(0),...,A(N) should occupy A(0). This value can be found by computing the minimum of A(0),...,A(N) and interchanging the contents of A(0) with the element of A where it was found. Now, we have a simpler, but similar situation. we must find the minimum of A(1),...,A(N) and again perform the appropriate interchange. The process can be continued until we have processed A(N-1) and A(N).

The first attempt at such an algorithm might be

```
DO I = 0 TO N - 1;

compute the index of the smallest
element of A(I), A(I+1),..., A(N) and
store that index into J;

interchange A(I) with A(J);
END ;
```

The interchange operation is particularly straightforward.

```
/* interchange A(I) with A(J) */

  TEMP = A(I);
  A(I) = A(J);
  A(J) = TEMP;
```

Now, to compute the index of the smallest element of A(I) through A(N), we write

```
/* assign J the value I, as a tentative index /*

    J = I;
    DO K = I + 1 TO N;
    IF A(K) < A(J) THEN J = K ;
    END ;
```

Write this sorting program in PLAGO and run it! ! !

There are many ways of sorting a sequence of values. Your recitation instructor will undoubtedly mention several others. Be certain that you understand the above program.

Exercises:

(1) Compute the mean, median and standard deviation of A(0)...A(N).

(2) Compute the sum of A(0)... A(N)

(3) Compute the greatest common divisor of A(0),...,A(N).

(4) Compute the least common multiple of A(0),...,A(N).

(5) Rewrite the following programs so that data is processed from the contents of arrays rather than as input values

a) The Birthday Problem

b) The GCD Problem

● * *

Arrays came in infinitely flavors, depending upon their dimensionality. Thus, the array A above was a one-dimensional array. A two-dimensional array, B, might be declared as

```
DECLARE B(25,30) FLOAT ;
```

Such an array can be visualized as a two dimensional table of simple variables having 25 rows and 38 columns. Thus B(5,17) names the item in the fifth row and the seventeenth column. You should gain a mastery of systematically storing and retrieving values from such arrays. As an example of such a computation, suppose that the variable N contains an integer value such that 1<=N<=25 and that the elements in the first N rows of the first N columns contain values.

> Compute the sum of the elements A(l,l), A(2,2),..., A(N,N) and store the result into MDS. Also compute the sum of the elements A(1,N), A(2,N-1),...,A(N,1) and store this value into SDS;

Clearly, we have

```
Set MDS and SDS to 0;

For each row of B, say I (i=l,2,..,N) add
to MDS B(I,I) and add to SDS the value
B(I,N+1-I);
```

Hence the fragment

```
MDS, SDS - 0 ;

DO I * 1 TO N;
MDS = MDS +B(I,I);
SDS = SDS + B(N + 1 - I);
```

Exercises:

(1) Write programs which input (output) values to (from) variously dimensioned arrays.

(2) Write programs, which test square arrays for

a) symmetry

b) diagonal dominance

c) whether or not the array is a Latin Square or a magic square.

Some Words About Recursion

In EWD316, Dijkstra devotes a chapter to discussing several ways of writing programs which correspond to recurrence relations. Skim the chapter before you read the text below.

There are many recursive definitions which arise in mathematics. A definition of N-factorial can be expressed as:

$0! = 1$
$N! = N * (N - 1)!$ where N is an integer greater than 0

The Fibonacci sequence from Chapter 1 of the book by Forsythe et al. can also be defined recursively

$F_1 = 1$

$F_2 = 1$

$F_N = F_{N-1} + F_{N-2}$ for $N > 2$

Recursive definitions occur quite frequently in numerical analysis. One such definition defines the Chebyschev polynomials of the first kind (bear with me please!) They are:

$T_0(X) = 1$

$T_1(X) = X$

$T_N(X) = 2 * X * T_{N-1}(X) - T_{N-2}(X), N > 1$

Now, the obvious question is:

How can recursive definitions be used to write programs.

The answer is frequently quite simple. Since PLAGO allows procedures to call themselves, recursive procedures can be written by following these steps:

(1) Explicitly test for the cases where a closed form result can be returned and return the value as appropriate.

(2) For all the remaining cases, call the procedure recursively with the appropriate arguments.

Thus the procedure T which computes the value of the N-th Chebyschev polynomial at X can be written as:

```
T.. PROCEDURE ( X, N ) RETURNS ( FLOAT ) ,.
    DECLARE X FLOAT, N FIXED ,.

    IF N = 0 THEN RETURN ( 1.0 ) ,.

    IF N = 1 THEN RETURN ( X ) ,.

    RETURN ( 2 * X * T ( X, N - 1 ) - T ( X, N - 2 ) ) ,.
    END T ,.
```

Study the above procedure carefully.

To help clarify some of these ideas, consider the following exercise.

On the following few pages, are lots of copies of the above procedure. Cut them out, and staple them together so that you have a booklet of identical pages, each page containing just one copy of the procedure. (That's right, cut out the next few pages and staple them together!) Notice that at the top of each are two boxes, one labelled N and another labelled X. These boxes will contain appropriate values for N and X. Now simulate the execution of T where X equals 4 and N equals 4.

Do this by first writing the above values in the boxes at the top of the first page of your booklet. Simulate the procedure. Clearly, in order to return the required value for T, other evaluations of T must be made. Do this by marking the function call that will be made (just put an arrow under T(X,N-1)) and then turn to the next page where there is a new copy of T. Insert the appropriate values for N and X in the boxes (N = 3, X = 4) and execute this procedure. Continue this process until a procedure can be executed to completion. In this case, simply write the value to be returned in the upper right corner of the sheet; TEAR IT OUT (That's right!); and flip to the immediately preceding page and write the value you wrote in the corner of the sheet that was torn out of the book beneath the marker you left behind. That's the value of the marked call! Continue evaluation by flipping to a clean copy of T or going back to a previous copy of T. The whole process terminates when the first page has a value in the upper right corner.

Can you think of another way of simulating a recursive procedure? (Hint: Consider stacking the values of X and N, similar to the way values were stacked in the discussion of arithmetic expressions.)

Recursive procedures have the property that they are usually short and concisely represent a computation. They also have the property of executing rather slowly (there are notable exceptions to that observation, however, cf. The Marriage Problem). Therefore, it is frequently, to your advantage to try to represent recursive algorithms as non-recursive ones, AFTER the recursive algorithm seems to behave properly. Several of the Problems address exactly this issue.

```
T.. PROCEDURE ( X, N ) RECURSIVE RETURNS ( FLOAT ) ,.
  DECLARE X FLOAT, N FIXED ,.
                                                  N [          ]
  IF N = 0 THEN RETURN ( 1.0 ) ,.
                                                  X [          ]
  IF N = 1 THEN RETURN ( X ) ,.

  RETURN ( 2 * X * T ( X, N - 1 ) - T ( X, N - 2 ) ) ,.


  END T ,.
```

```
T.. PROCEDURE ( X, N ) RECURSIVE RETURNS ( FLOAT ) ,.
  DECLARE X FLOAT, N FIXED ,.

  IF N = 0 THEN RETURN ( 1.0 ) ,.                 N [          ]

  IF N = 1 THEN RETURN ( X ) ,.                   X [          ]

  RETURN ( 2 * X * T ( X, N - 1 ) - T ( X, N - 2 ) ) ,.


  END T ,.
```

```
T.. PROCEDURE ( X, N ) RECURSIVE RETURNS ( FLOAT ) ,.
  DECLARE X FLOAT, N FIXED ,.
                                                  N [          ]
  IF N = 0 THEN RETURN ( 1.0 ) ,.
                                                  X [          ]
  IF N = 1 THEN RETURN ( X ) ,.

  RETURN ( 2 * X * T ( X, N - 1 ) - T ( X, N - 2 ) ) ,.


  END T ,.
```

```
T.. PROCEDURE ( X, N ) RECURSIVE RETURNS ( FLOAT ) ,.
   DECLARE X FLOAT, N FIXED ,.

   IF N = 0 THEN RETURN ( 1.0 ) ,.          N [          ]

   IF N = 1 THEN RETURN ( X ) ,.            X [          ]

   RETURN ( 2 * X * T ( X, N - 1 ) - T ( X, N - 2 ) ) ,.

   END T ,.
```

```
T.. PROCEDURE ( X, N ) RECURSIVE RETURNS ( FLOAT ) ,.
   DECLARE X FLOAT, N FIXED ,.

   IF N = 0 THEN RETURN ( 1.0 ) ,.          N [          ]

   IF N = 1 THEN RETURN ( X ) ,.            X [          ]

   RETURN ( 2 * X * T ( X, N - 1 ) - T ( X, N - 2 ) ) ,.

   END T ,.
```

```
T.. PROCEDURE ( X, N ) RECURSIVE RETURNS ( FLOAT ) ,.
   DECLARE X FLOAT, N FIXED ,.
                                            N [          ]
   IF N = 0 THEN RETURN ( 1.0 ) ,.
                                            X [          ]
   IF N = 1 THEN RETURN ( X ) ,.

   RETURN ( 2 * X * T ( X, N - 1 ) - T ( X, N - 2 ) ) ,.

   END T ,.
```

```
T.. PROCEDURE ( X, N ) RECURSIVE RETURNS ( FLOAT ) ,.
    DECLARE X FLOAT, N FIXED ,.                          N
    IF N = 0 THEN RETURN ( 1.0 ) ,.
                                                         X
    IF N = 1 THEN RETURN ( X ) ,.

    RETURN ( 2 * X * T ( X, N - 1 ) - T ( X, N - 2 ) ) ,.


    END T ,.
```

```
T.. PROCEDURE ( X, N ) RECURSIVE RETURNS ( FLOAT ) ,.
    DECLARE X FLOAT, N FIXED ,.

    IF N = 0 THEN RETURN ( 1.0 ) ,.                      N

    IF N = 1 THEN RETURN ( X ) ,.                        X

    RETURN ( 2 * X * T ( X, N - 1 ) - T ( X, N - 2 ) ) ,.


    END T ,.
```

```
T.. PROCEDURE ( X, N ) RECURSIVE RETURNS ( FLOAT ) ,.
    DECLARE X FLOAT, N FIXED ,.                          N
    IF N = 0 THEN RETURN ( 1.0 ) ,.
                                                         X
    IF N = 1 THEN RETURN ( X ) ,.

    RETURN ( 2 * X * T ( X, N - 1 ) - T ( X, N - 2 ) ) ,.


    END T ,.
```

```
T.. PROCEDURE ( X, N ) RECURSIVE RETURNS ( FLOAT ) ,.
    DECLARE X FLOAT, N FIXED ,.
                                                  N
    IF N = 0 THEN RETURN ( 1.0 ) ,.
                                                  X
    IF N = 1 THEN RETURN ( X ) ,.

    RETURN ( 2 * X * T ( X, N - 1 ) - T ( X, N - 2 ) ) ,.


    END T ,.
```

```
T.. PROCEDURE ( X, N ) RECURSIVE RETURNS ( FLOAT ) ,.
    DECLARE X FLOAT, N FIXED ,.

    IF N = 0 THEN RETURN ( 1.0 ) ,.               N

    IF N = 1 THEN RETURN ( X ) ,.                 X

    RETURN ( 2 * X * T ( X, N - 1 ) - T ( X, N - 2 ) ) ,.


    END T ,.
```

```
T.. PROCEDURE ( X, N ) RECURSIVE RETURNS ( FLOAT ) ,.
    DECLARE X FLOAT, N FIXED ,.
                                                  N
    IF N = 0 THEN RETURN ( 1.0 ) ,.
                                                  X
    IF N = 1 THEN RETURN ( X ) ,.

    RETURN ( 2 * X * T ( X, N - 1 ) - T ( X, N - 2 ) ) ,.


    END T ,.
```

```
T.. PROCEDURE ( X, N ) RECURSIVE RETURNS ( FLOAT ) ,.
    DECLARE X FLOAT, N FIXED ,.
                                                  N [        ]
    IF N = 0 THEN RETURN ( 1.0 ) ,.
                                                  X [        ]
    IF N = 1 THEN RETURN ( X ) ,.

    RETURN ( 2 * X * T ( X, N - 1 ) - T ( X, N - 2 ) ) ,.


    END T ,.
```

```
T.. PROCEDURE ( X, N ) RECURSIVE RETURNS ( FLOAT ) ,.
    DECLARE X FLOAT, N FIXED ,.

    IF N = 0 THEN RETURN ( 1.0 ) ,.               N [        ]

    IF N = 1 THEN RETURN ( X ) ,.                 X [        ]

    RETURN ( 2 * X * T ( X, N - 1 ) - T ( X, N - 2 ) ) ,.


    END T ,.
```

```
T.. PROCEDURE ( X, N ) RECURSIVE RETURNS ( FLOAT ) ,.
    DECLARE X FLOAT, N FIXED ,.
                                                  N [        ]
    IF N = 0 THEN RETURN ( 1.0 ) ,.
                                                  X [        ]
    IF N = 1 THEN RETURN ( X ) ,.

    RETURN ( 2 * X * T ( X, N - 1 ) - T ( X, N - 2 ) ) ,.


    END T ,.
```

Preface to the Problems

The programs which accompany the fallowing problems were all run as PLAGO programs. Each compiled and executed correctly. Hopefully, these programs will serve as models as well as objects subject to criticism. Several of the problems make reference to an introductory text by Forsythe, Organick, Keenan, and Stenberg. The book:

"Computer Science: A First Course"

is on reserve in the library.

* * •

Because of the limited character set which can be printed by the line printer from which you will receive listings of your programs, the following PL/I characters are printed as indicated

PL/I	Printer
	NE
>	GT
<	LT
	NG
	NL
<=	LE
>=	GE
	NOT
I	OR
II	CAT

Always punch the characters appearing in the left-hand column, NEVER the ones in the right-hand column.

* « «

One minor difficulty which you might encounter has to do with the programming notation used by Dijkstra in EWD316 and the notation required by PL/I. These difficulties arise because, in many cases, both use the same notation to mean slightly different things. The most important of these are listed below.

(1) The assignment operator in PL/I is V and not V. However, statements having multiple left parts in PL/I are written with the left parts separated by commas

I, J, K : 8; means I « J » K » B;

(2) The while clause which Dijkstra uses is of the form

```
    while ℬℰ do
```

in PL/I, its equivalent is

```
    DO WHILE (ℬℰ);
```

(3) The repeat statement

```
    repeat 𝒮 until ℬℰ;
```

has only several messy equivalents in PL/I. One such equivalent is a form of the DO statement which uses a variable called REPEAT

```
1       DO REPEAT = 0, 0 BY 0 WHILE (¬ℬℰ );
```

Another, more straightforward equivalent is

```
    R = 1;
    DO WHILE ( R|¬ℬℰ);
     .
     .
     .
    R = 0;
    END;
```

Study both of these forms and find several of your ou situations where either of the above will fail?

(4) Dijkstra uses begin and end to parenthesize statements. In PL/I, DO; and END; parenthesize statements and BEGIN; and END; delimit blocks!

Computing the Greatest Common Divisor

Why I've included this problem:

It provides an example of some of the difficulties and shows some of the techniques one encounters when transforming a structured description into a running program.

The algorithm first described can easily be made a part of a program which computes the GCD of a sequence of pairs of positive integers, thereby providing a simple example of a complete program, including all the input/output statements.

The Problem:

On page 37 of EWD316 is a program which computes the greatest common divisor of two positive integers. Suppose we wish to extend this program so that it computes the greatest common divisor of arbitrarily many pairs of positive integers. One way of doing this involves punching the sequence of pairs into data cards. We can terminate the sequence by following the last pair of integers by a pair of zeros. Hence, an algorithm which solves the problem might be.

```
input values for A and B;

while A is not equal to 0 do
    begin

    print the values of A and B;

    compute the GCD of A and B and
        leave the result in GCD;

    print the value of GCD;

    input values for A and B;
    end ;
```

A PLAGO program which is equivalent to this description is

```
G.. PROCEDURE OPTIONS ( MAIN ) ;.

/* PRINT THE VALUES OF A SEQUENCE OF PAIRS OF POSITIVE INTEGERS */
/* AND THEIR GREATEST COMMON DIVISORS.  THE INPUT WILL          */
/* BE TERMINATED BY A PAIR OF ZEROS                             */

DECLARE ( A, B, GCD ) FIXED ;.

/* INPUT VALUES FOR A AND B */
```

```
GET LIST  (A, B ) ;
  DO WHILE (A NE 0 & B NE 0 ) ;
/* PRINT THE VALUES OF A AND B */
  PUT SKIP LIST ('A = ', A, ' / B = ', B ) ;
/* COMPUTE THE GCD OF A AND B AND LEAVE THE RESULT IN GCD */
    DO WHILE ( A NE B ) ;
      DO WHILE (A GT B ) ;
      A = A - B ;
      END ;

      DO WHILE ( B GT A ) ;
      B = B - A ;
      END ;
    END ;
  GCD = A ;
/* PRINT THE VALUE OF THE GCD OF A AND B */
  PUT LIST ( ' GCD = ', GCD ) ;
  GET LIST ( A, B ) ;
  END ;
END ;
```

Exercise:

(1) Write and run a PLAGO program which prints the values of a sequence of pairs of positive integers and their greatest common divisors and their smallest common multiples. The input should be terminated by a pair of zeros. Use the program on page 41 of EWD316. Your solution should include the set of stepwise refinements which led to the program.

(2) PLAGO has a special built-in function called MOD which does the following

MOD(se1, se2) has the value of the remainder of the division se1/se2

For example

MOD(20, 7) equals 6;
MOD(2, 6) equals 2

If you are allowed to use only the MOD function and no other arithmetic operations, how would the GCD program change? Rewrite it using only the MOD function (comparisons of variables are still allowed, but not of more complicated expressions!)

Solving Quadratic Equations

Why I've included this problem:

> Little mathematical background is needed to understand the problem. Hence, the development can concentrate on programming issues.

The Problem

The equation

```
A * X ** 2 + B * X + C = 0
```

can be solved, when A is not equal to 0 by

$$\frac{-B \text{ + or } - \sqrt{B * B - 4 * A * C}}{2 * A}$$

We wish to write a program which will accep , as its input, values for A, B, and C, and produce, as output, the values of the root or roots of the equation. Thus, a first description of the solution might be

```
input values for A, B, and C;

output values of A, B, and C;

solve A * X ** 2 + B * X + C = 0,
      and output the values of the
      roots along with the case
      which was solved;
```

Several situations arise, however, in attempting to solve the equation. First, if A is not equal to 0, the formula applies. If not, and B is not equal to 0, then the equation is linear in X and has a root which is -C/B. If B = 0 and C is not equal to 0 then no equation is represented. We might wish to print some kind of error message to accompany this case. Finally, if A = 0 and B = 0 and C = 0, an identity is represented. Again, a message might be appropriate as part of the output.

A refinement of the third statement might be

```
/* solve A * X ** 2 + B * X + C = 0 */

    if A not equal to 0 then
         begin
         solve the quadratic using the formula;
         end
      else if B not equal to 0 then
             begin
         solve the linear equation;
         end
       else if C not equal to 0 then
            begin
            print a message saying that no
              equation is represented;
            end
          else
            begin
            print a message saying that an identity
               is represented (0=0);
            end
```

The quadratic formula may be evaluated by observing that if

$$B * B - 4 * A * C \begin{cases} = 0, \text{ there is one real root} \\ > 0, \text{ there are 2 real roots} \\ < 0, \text{ there are 2 complex roots} \end{cases}$$

Hence the final program is

```
QUAD.. PROCEDURE OPTIONS( MAIN ) ,.
DECLARE ( A, B, C, DISC, SQD ) FLOAT ,.

/* INPUT VALUES FOR A, B, AND C */

GET LIST ( A, B, C ) ,.

/* OUTPUT VALUES OF A, B, AND C */

PUT SKIP LIST ( ' A = ', A, ' / B = ', B, ' / C = ', C ) ,.

/* SOLVE A * X ** 2 + B * X + C = 0 AND OUTPUT THE VALUES OF THE */
/* ROOTS WITH THE CASE WHICH WAS SOLVED                          */

IF A NE 0 THEN
   DO,.

/* SOLVE THE QUADRATIC WITH THE FORMULA */

   DISC = B * B - 4 * A * C ,.
   IF DISC = 0 THEN

/* THERE IS ONE REAL ROOT */

      PUT SKIP LIST (' THERE IS ONE REAL ROOT WHICH EQUALS ',-B/2/A),.
```

```
    IF DISC GT 0 THEN

/* THERE ARE TWO REAL ROOTS */

      DO,.
      SQD = SQRT ( DISC ) ,.
      PUT SKIP LIST (' THERE ARE TWO REAL ROOTS,', (-B + SQD)/2/A,
         ' AND ', -(B + SQD ) / 2 / A ) ,.
      END ,.
  ELSE

/* THERE ARE TWO COMPLEX ROOTS */

      DO ,.
      SQD = SQRT ( - DISC ) ,.
      PUT SKIP LIST (' THERE ARE TWO COMPLEX ROOTS,',
         -B/2/A, '+', SQD/2/A,' * I ', ' AND ',
         -B/2/A, ' - ', SQD/2/A, ' * I' ) ,.
      END,.
    END ,.
  ELSE IF B NE 0 THEN
    PUT SKIP LIST (' THERE IS ONE REAL ROOT - LINEAR CASE ', -C / B),.
  ELSE
    IF C NE 0 THEN
      PUT SKIP LIST ( '//// NO EQUATION IS REPRESENTED ////') ,.
    ELSE
      PUT SKIP LIST ( ' THE IDENTITY 0 = 0 IS REPRESENTED ') ,.
  END ,.
```

A Birthday Problem

Why I've included this problem:

Its analysis is straightforward.

The computations in the final program must be arranged so that overflows do not occur at intermediate stages of computation.

The Problem:

Suppose that K persons are gathered in a room. What is the probability that at least two of the persons were born on the same day of the year? (Ignore the possibility of anyone being born on February 29)

The problem can be analyzed by noting that the answer equals

$$1 - \left\{ \begin{array}{l} \text{the probability that no two} \\ \text{persons in the room were born} \\ \text{on the same day of the year} \end{array} \right\}$$

The quantity in braces is now just the number of ways K persons can have different birthdays divided by the total number of ways K persons can have birthdays, i.e.

$$\frac{365 * 364 * \ldots * (365 - K + 1)}{365 ** K}$$

Note: Those students worried about the relevance of this problem may wish to consider the solution to the following:

An electronic assembly contains K components, each of which will fail sometime during the next N time periods. The assembly will continue to operate if only single components fail in a time period, but will fail if more than one component fails in a time period. What is the probability that the assembly will fail? Let N be 365 to be definite!

The solution to this problem can be extended to allow it to compute a sequence of probabilities, i.e. we wish to print the values of N positive K's (the number of people in the room) and for each K, the probability that at least two of them were born on the same day of the year. The values of K are to be read from data cards. Preceding the first value for K is a positive integer, N, corresponding to the number of times K is to be assigned a new value, implying a new computation of the probability.

The first stage in the development might be

```
input a value to N;
```

```
while N > B dfl
   begin
   input a value to K;

   output the value oi K;

   compute the value of the probability that
      at least two people, among K, were born
      on the same da/ of the year.  Store this
      value into PROB;

   output the value of PROB;

   N := N - 1;
   end
```

The details of developing ail the parts of the design, except the computation of PHOB are straightforward. They appear in the final program. However, the task of computing PROB requires more analysis.

Several cases are apparent. First, if the value of K is less than 2, the probability of two people being barn on the same day of the year is, of course, zero. Further, if there are more than 365 people in the room, the probability that at least two were born on the same day of the year is 1. In the remaining cases, the formula can be calculated. Thus, we have

```
/* compute the probability for K and store it into PROB */

        ji K < 2 then PROB := 8
                 else if K > 365 then PROB := 1
                                 else
                                   /* compute the formula */
```

The formula can now be refined as follows. We select DEN to represent the value of the denominator and NUM to represent the value of the numerator. Both can initially be set to 1 to get

```
        NUM :« DEN := 1;

        I » 1;

        while I <= K do
            begin
            NUM := NUM * (366 - I);
            DEN    DEN » 365;
            1    I . 1;
            end ;

        PROB := 1 - NUM / DEN;
```

The final program is now

```
BDAY.. PROCEDURE OPTIONS ( MAIN ) ,.

/* READ A VALUE INTO N, INDICATING THE NUMBER OF TIMES A VALUE IS */
/* TO BE READ INTO K. PRINT EACH K ALONG WITH THE PROBABILITY THAT*/
/* AT LEAST TWO OF K PEOPLE IN A ROOM WERE BORN ON THE SAME DAY   */
/* OF THE YEAR.                                                   */

DECLARE ( I, N, K ) FIXED ,.
DECLARE ( NUM, DEN, PROB ) FLOAT ,.

/* INPUT A VALUE FOR N */

GET LIST ( N ) ,.
  DO WHILE ( N GT 0 ) ,.

/* INPUT A VALUE FOR K */

  GET LIST ( K ) ,.

/* OUTPUT VALUE OF K */

  PUT SKIP LIST ( ' K = ', K ) ,.

/* COMPUTE THE PROBABILITY FOR K AND STORE THE RESULY
/* COMPUTE THE PROBABILITY FOR K AND STORE THE RESULT IN PROB */

  IF K LT 2 THEN PROB = 0 ,.
    ELSE IF K GT 365 THEN PROB = 1 ,.
      ELSE
        DO ,.
        NUM, DEN = 1 ,.
        I = 1 ,.
          DO WHILE ( I LE K ) ,.
          NUM = NUM * ( 366 - I ) ,.
          DEN = DEN * 365 ,.
          I = I + 1 ,.
          END ,.
        PROB = 1 - NUM / DEN ,.
        END ,.
  PUT SKIP LIST ( ' PROB = ', PROB ) ,.
  N = N - 1 ,.
  END ,.
END ,.
```

```
          K =                                    2
          PROB =                       2.73973E-03
          K =                                    7
          PROB =                       5.62357E-02
          K =                                   20
          PROB =                       4.11438E-01
          K =                                   30

         CONDITION 'OVERFLOW' SIGNALLED IN STATEMENT  15

         CONDITION 'ERROR' SIGNALLED IN STATEMENT  15

         CONDITION 'FINISH' SIGNALLED IN STATEMENT  15.
```

Unfortunately, this PLAGO program will fail for several values of K. The reason for this is that the finite capacity of a storage cell is exceeded during an intermediate calculation (EWD316, p.26). This explains the peculiar message in the output. It's not difficult to see that if K is, say, 75, the value of the denominator exceeds 10**150, which exceeds the default magnitude of a FLOAT variable.

A much better way of performing the calculations would be to initialize PROB to 1 and within the loop compute:

```
PROB := PROB * ( 366 - I ) / 365
```

This assures us that intermediate calculations will not lead to results which are extremely large.

Exercise:

(1) Modify the program using the above suggestion. Could the suggestion lead to other kinds of difficulties?

(2) Consider the following simple problem:

> Suppose you wish to compute the distance between two points in a plane. Let the coordinates of the first point be represented in the variables X1 and Y1 and those of the second in X2 and Y2. The formula
>
> ```
> (X1 - X2) ** 2 + (Y1 - Y2) ** 2
> ```
>
> computes the value we want. Now suppose that you are guaranteed that the distance between the two points will not raise the overflow condition. How can you guarantee that no intermediate calculation in the above formula - or a modification of it - will raise the overflow condition?

Develop a PLAGO program which computes the distance between pairs of points. The input should contain a value for N, as the first value of the input, followed by N groups of four values, corresponding to the coordinates of two points. The program should output the values of these coordinate pairs along with the distance which separates the two points.

(3) Modify the program from exercise (2) so that the value of the shortest(longest) distance is printed at the end of the output.

A Nest of Squares

Why I've included this problem:

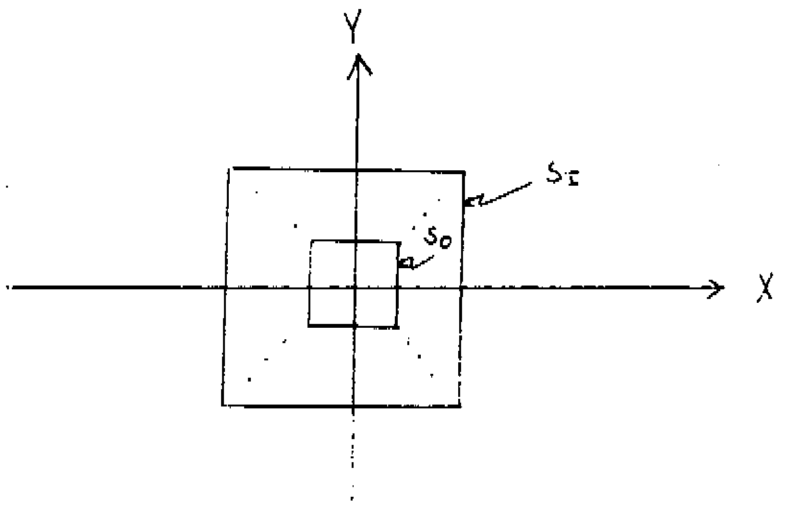
> This problem shows how an algorithm can be transformed into a lower echelon algorithm just by recognizing a simple property.

The Problem:

Suppose that a family of squares, S(0), S(1), ... , S(I), ... is defined so that the area of square S(I) equals

(I + 1) * A, where A is positive and real.

Suppose further that this family of squares is centered at the origin of a two-dimensional coordinate system with sides parallel to the X and Y axes. For example:



Now imagine that the variables X and Y define the respective X and Y coordinates of some point. What is the index of the smallest square which contains the point (X,Y)?

For example, if A is 1, X is 4, and Y is 3, then the index of the smallest square containing (4,3) is 63 - S(63) is the smallest square containing (4,3). (convince yourself that this is true before going on)

This problem can be analyzed in several ways. One way is to notice that since each square is symmetric about the X and Y axes, the smallest square in our family containing (X,Y) also contains the smallest square centered at the origin with sides parallel to the axes, and with (X,Y) on its boundary. Hence the area of each square in the family (starting with the smallest) can be compared with the area of the square with (X,Y) on its boundary - call this square S. The first square whose area is greater than or equal to the area of S is the square whose index answers our question.

More concisely, we might write:

```
ASQ←area of square with point (X,Y)
    on its boundary;
I ← 0 ;
while area of S(I) < ASQ compute I←I + 1 ;
```

```
INDEX.. PROCEDURE OPTIONS (MAIN) ,.
  DECLARE (X, Y, A, ASQ) FLOAT,  (I) FIXED ,.
/* GRAB SOME INPUT VALUES AND PRINT THEM */
  GET LIST (A, X, Y ) ,.
  PUT LIST ( 'A = ', A, 'X = ', X, 'Y = ', Y ) ,.
/* COMPUTE THE AREA OF THE SMALLEST SQUARE CONTAINING (X,Y) */
  ASQ = 4 * MAX( ABS (X), ABS (Y) )   ** 2 ,.
/* COMPUTE THE INDEX OF THE SMALLEST SQUARE CONTAINING (X, Y) */
  DO I = 1 BY 1 WHILE ( ASQ GT A * I) ,.
  END ,.
I = I - 1 ,.
/* I CONTAINS THE VALUE WE ARE AFTER... SO, PRINT IT */
  PUT LIST ('INDEX OF SMALLEST SQUARE CONTAINING (X,Y) IS ', I ) ,.
END INDEX ,.
```

The more intrepid analyst, however, might notice that there are infinitely many values of I for which this inequality holds:

(area of S) <= A * (I + 1)

Solving this for I yields

(area of S) / A - 1 <= I.

Clearly the left side can be computed. Therefore, if we can compute the value of the smallest integer which is greater than or equal to the left side, our question is again answered!

The following program does just this. Why? Think of some other ways of solving this problem.

```
INDEX.. PROCECURE OPTIONS (MAIN) ,.
   DECLARE (X, Y, A, ASQ) FLOAT,   (I) FIXED ,.
/* GRAB SOME INPUT VALUES AND PRINT THEM */
   GET LIST (A, X, Y ) ,.
   PUT LIST ( 'A = ', A, 'X = ', X, 'Y = ', Y ) ,.
/* COMPUTE THE AREA OF THE SMALLEST SQUARE CONTAINING (X,Y) */
   I = CEIL ( ( 4 * MAX( ABS(X), ABS(Y) ) ** 2 ) / A - 1 ) ,.
/* I CONTAINS THE VALUE WE ARE AFTER... SO, PRINT IT */
   PUT LIST ('INDEX OF SMALLEST SQUARE CONTAINING (X,Y) IS ', I ) ,.
END INDEX ,.
```

Evaluating Arithmetic Expressions

A quiz similar to the following was given during a 15-100 lecture. Try it. Don't spend more than 10 minutes.

The variables in the following expressions have the values indicated in the table:

A	B	C	E	I	J	K
3	4	7	3	1	2	10

Evaluate each of the following expressions:

Expression 1:

```
A + B + C / I / I / I * K - B * C
```

Expression 2:

```
A * B + C - ( E + K / 5 ) ** ( 3 - I ) * ( J - 2 * ( C + A ) )
```

Expression 3:

```
( A + B + C / I / I / I * K - B * C ) * B + C - ( E + K / 5 ) ** ( 3 - I )
* ( J - 2 * ( C + A ) ) - K + J - E * ( A - B * K * ( C - E / I ) - 4 * ( I
+ K - C ) ) + ( C - 5 + K / J / I - 2 ) ** 1 + B - J / ( C - E - J ) + A *
K - B * ( ( ( ( ( J + C ) * ( K - 4 ) / J - I ) * C - I ) ) ) * E + A )
```

The results of the quiz are easy to describe. Almost everyone evaluated the first expression correctly; about half the students evaluated the second expression correctly; and no one evaluated the third expression correctly! WHY. If you examine the three expressions, you should note that the only essential difference between them is their lengths. All the arithmetic operations are trivial. Probably the reason students had so much trouble with the last expression was because they didn't have a very careful bookkeeping system which would tell them when to perform arithmetic and on what to perform it. The methods described in your textbook I find rather clumsy (you may not). Therefore, I have written a flowchart which evaluates arithmetic expressions by scanning them from left to right without ever re-scanning any pact of the expression.

The flowchart which follows - an informal but precise one - does this by systematically postponing arithmetic operations until they can be performed. This is accomplished with the aid of an OPERATOR STACK and an OPERAND STACK.

Before you proceed, take a look at the flowchart. Pay special attention to the comments.

Let me demonstrate the flowchart by using it to evaluate the expression:

A * B + C- (E . K/ 5) * * (3 - I) * (J - 2 * (C . A))

where the variables have the values tabulated below

B	C J	E	i	J	K
4	7:	3	i	2	18

The algorithm begins by inserting the symbol - to the right of the rightmost symbol in the arithmetic expression. This symbol - sometimes called a "right terminator" or "right turnstile" - simply signals the end of the arithmetic expression. Before proceeding, arm youself with a bunch of small slips of paper. Make sure that each slip can fit inside the labelled squares on the page following the flowchart. Next, place some kind of pointer (a pencil mark will do) beneath the leftmost symbol in the expression. By symbol we mean a variable name or constant or arithmetic operator or parenthesis.

Now, determine whether the symbol is a variable name or a constant. In the example, the symbol is a variable name, A. So, "push" the value of the variable name onto the OPERAND STACK. This amounts to simply jotting the value of A on a slip of paper and placing this slip on top of the pile (possibly empty) of slips inside the square labelled OPERAND STACK. Next, advance the pointer one symbol to the right and follow the flowchart until you find the test box which inquires as to the PRECEDENCE of the newly scanned operator. This box asks whether the precedence of the scanned operator is greater than the precedence of the operator at the top of the OPERATOR STACK. By convention, we say that an empty stack and a left-parenthesis have lower precedence than all the operators. Hence we copy the symbol V onto a slip of paper and "push" it onto the OPERATOR STACK. Again, move the pointer one symbol to the right; scan B; push its value onto the OPERAND STACK; move the pointer one symbol to the right; and scan V. Here, note that V has lower precedence than V (which is the top of the OPERATOR STACK). Because of this circumstance, "pop" the top of the OPERATOR STACK to OP, i.e. move the slip on top of the OPERATOR STACK to the square called OP; "pop" the top of the OPERAND STACK to ROP; and "pop" the top of the OPERAND STACK to LOP. Next, perform the arithmetic operation "OP" on "LOP" and "ROP" and write the result on a new slip of paper. Push this value onto the OPERAND STACK and throw away the slips in OP, ROP, and LOP.

What we have just done has been to compute the product of A and B, with the result now on the OPERAND STACK. Now compare the precedence of the scanned symbol with the precedence of the symbol at the top of the OPERATOR STACK. Again since the OPERATOR STACK is empty, simply push the '+' onto the OPERATOR STACK.

The Table which follows is a sequence of "snapshots" describing the process by which the expression is evaluated. Note particularly how parenthesized sub-expressions are handled! Observe that when the flowchart stops that the value of the expression is the single value left in the OPERAND STACK! Don't let yourself get bogged down. The flowchart is straightforward but somewhat tedious. It might be helpful for you to look at the flowchart again before proceeding.

Snapshots of the Evaluation Process for

A * B + C - (E + K / 5) ** (3 - I) * (J - 2 * (C + A))

where

A	B	C	E	I	J	K
3	4	7	3	1	2	18

Note that the top of the OPERAND STACK and the top of the OPERATOR STACK is always the leftmost symbol in the appropriate column.

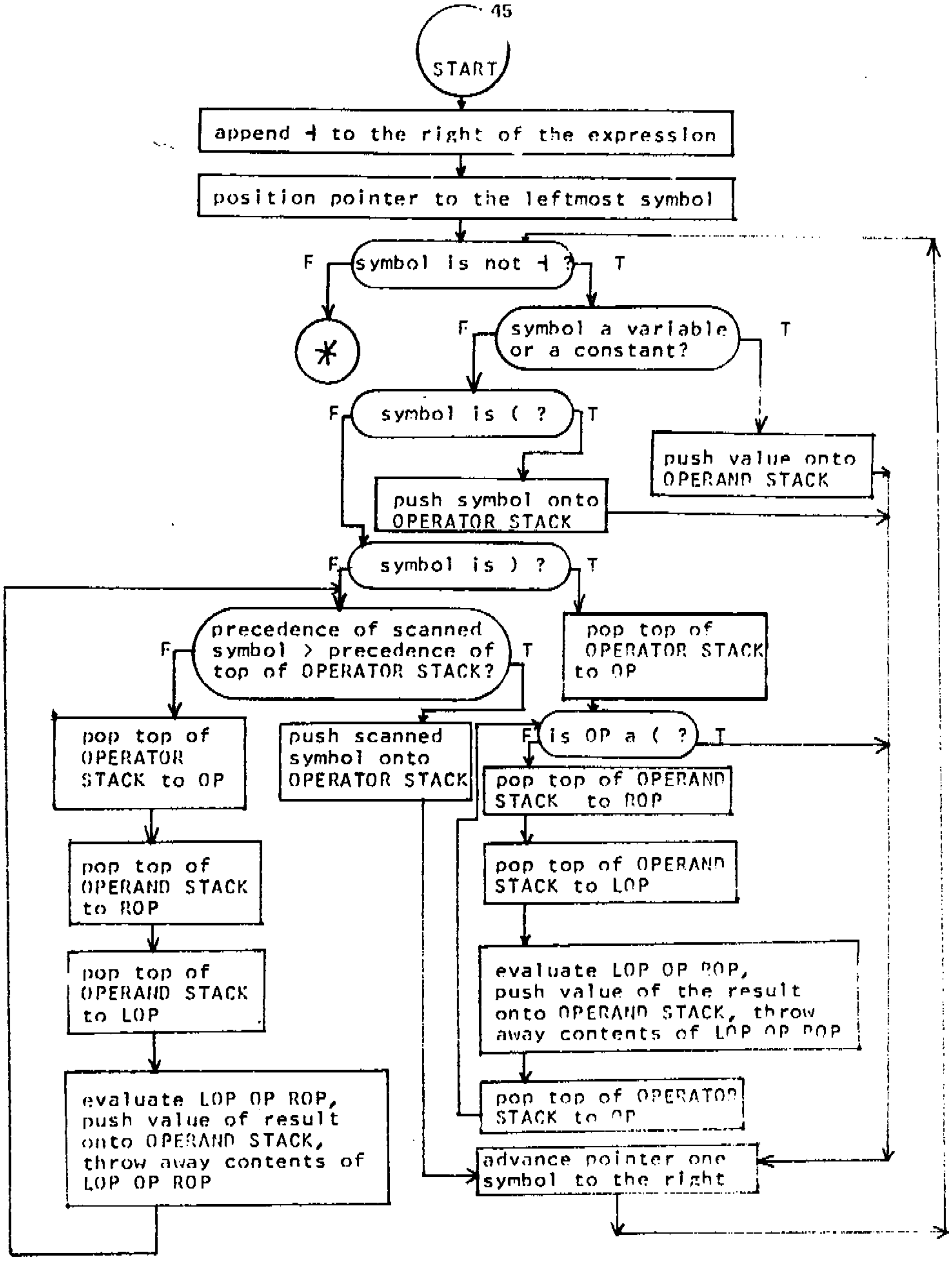
Scanned Symbol	LOP	OP	ROP	OPERAND STACK	OPERATOR STACK
A				3	
*				3	*
B				4 3	*
+	3	*	4		
+				12	+
C				7 12	+
-	12	+	7		
-				19	-
(				19	(-
E				3 19	(-
+				3 19	+ (-

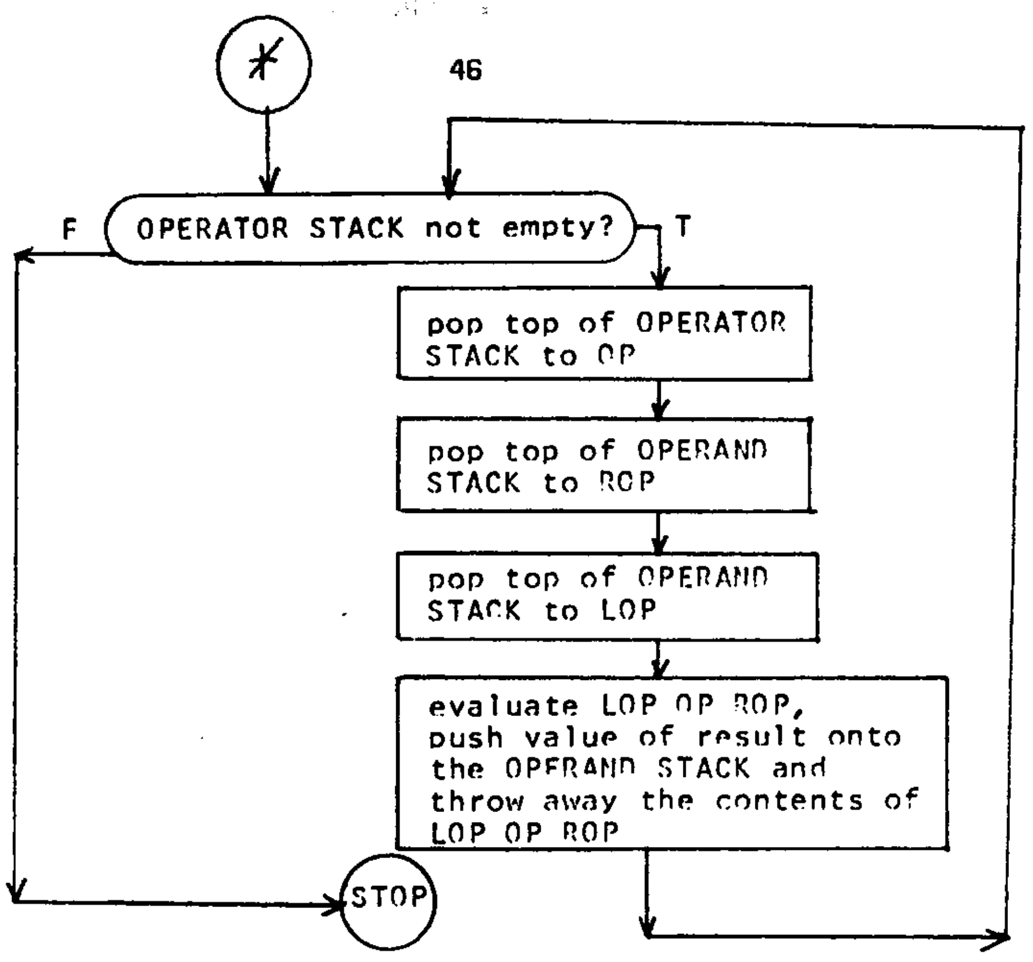
K		10 3 19	+ (-
/		10 3 19	/ + (-
5		5 10 3 19	/ + (-
)	10 / 5	3 19	+ (-
)		2 3 19	+ (-
)	3 + 2	19	(-
)		5 19	
)		5 19	-
**		5 19	** -
(		5 19	(** -
3		3 5 19	(** -
-		3 5 19	- (** -
I		1 3 5 19	- (** -
)	3 - 1	5 19	(** -
)		2 5 19	(** -
)		2 5 19	** -
*	5 ** 2	19	-
*		25 19	* -
(		25 19	(* -
J		2 25 19	(* -
-		2 25 19	- (* -
2		2 2 25 19	- (* -
*		2 2 25 19	* - (* -
(		2 2 25 19	(* - (* -
C		7 2 2 25 19	(* - (* -
+		7 2 2 25 19	+ (* - (* -
A		3 7 2 2 25 19	+ (* - (* -
)	7 + 3	2 2 25 19	(* - (* -
)		10 2 2 25 19	* - (* -

)	2	*	10	2 25 19	- (* -
)				20 2 25 19	- (* -
)	2	-	20	25 19	(* -
)				-18 25 19	(* -
)				-18 25 19	* -
⊢	25	*	-18	19	-
⊢				-450 19	-
⊢	19	-	-450		
⊢				469	

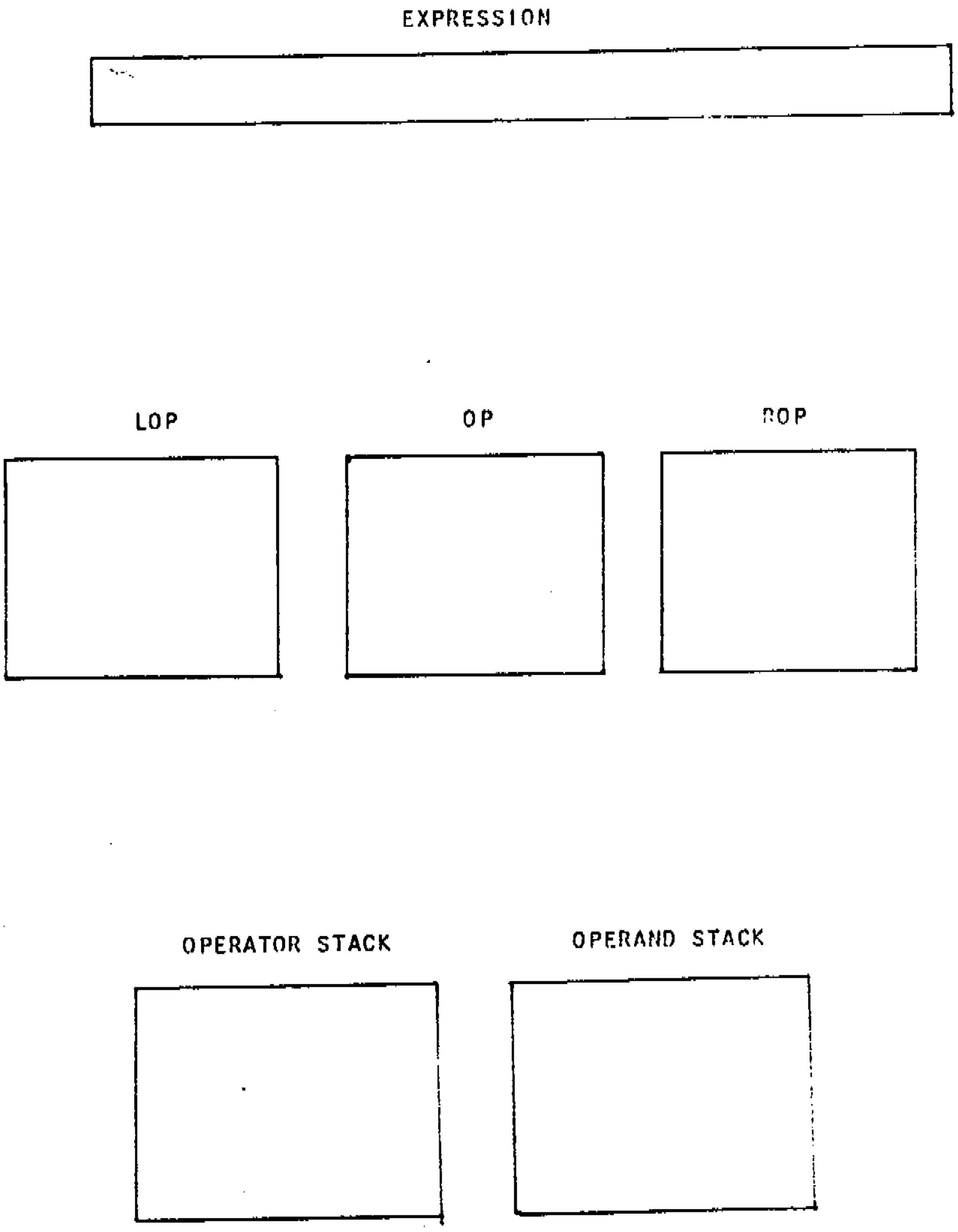
It should be clear that the flowchart doesn't behave properly for expressions containing unary '+' and '-' sign. Fix the flowchart to handle this case.

Modify the algorithm so that some special path and exit are followed in the event that the expression is discovered to be syntactically incorrect.


START
append ⊣ to the right of the expression
position pointer to the leftmost symbol
symbol is not ⊣ ?
F
T
*
symbol a variable or a constant?
F
T
push value onto OPERAND STACK
symbol is (?
F
T
push symbol onto OPERATOR STACK
symbol is) ?
F
T
pop top of OPERATOR STACK to OP
precedence of scanned symbol > precedence of top of OPERATOR STACK?
F
T
pop top of OPERATOR STACK to OP
push scanned symbol onto OPERATOR STACK
is OP a (?
F
T
pop top of OPERAND STACK to ROP
pop top of OPERAND STACK to LOP
pop top of OPERAND STACK to ROP
pop top of OPERAND STACK to LOP
evaluate LOP OP ROP, push value of the result onto OPERAND STACK, throw away contents of LOP OP ROP
evaluate LOP OP ROP, push value of result onto OPERAND STACK, throw away contents of LOP OP ROP
pop top of OPERATOR STACK to OP
advance pointer one symbol to the right



OPERATOR STACK not empty?
F
T
pop top of OPERATOR STACK to OP
pop top of OPERAND STACK to ROP
pop top of OPERAND STACK to LOP
evaluate LOP OP ROP, push value of result onto the OPERAND STACK and throw away the contents of LOP OP ROP
STOP


EXPRESSION
LOP
OP
ROP
OPERATOR STACK
OPERAND STACK

A Monotone Sequence

Why I've included this problem:

> Algorithms which solve this problem seem not to be immediately obvious, but can be developed in a step-wise way. I think that's a good property for a programming problem to have.
>
> The problem has some interesting generalizations.

The problem:

Put simply, if you have a linear array A, containing N different real values, find the length of the longest monotone increasing subsequence. The book by Forsythe et al. discusses this problem on pages 191-199. Read and understand that material before going on.

Write structured statements which correspond to the flowchart on page 199.

Now study the PLAGO program on the next page.

Rewrite it so that it computes the length of the longest monotone DECREASING sequence. Follow the notation and suggestions of exercise 4 on page 198.

Modify the program again so that it not only produces the length of the longest monotone increasing sequence, but also produces an instance of such a sequence. Exercise 4 on page 198 suggests a way of doing this. Create the subsequence by putting it into the first MAXINC elements of an array called MS.

Make sure you can prove the results in exercises 2 and 3 on page 198.

Can you think of other, more or less efficient, algorithms which solve the problem?

```
 PAIN.. PROCEDURE CPTIONS ( MAIN )
     DECLARE ( A(50), N ) FIXED

/» COMPUTE THE LENGTH OF THE LCNG6ST MONOTONE INCREASING SEQUENCE */
/* IN A(1)...A(N)                                                 */

 MCNSEQ.. PRCCECURE (A, N) RETURNS ( FIXED )
 CECLARE < J» K« A(N), 8(N),N, MAXINC I FIXEO

/* SET LENGTH CF LONGEST INITIAL SEQUENCE TO 1 */

      *AXINC = I
        CO J = 1 TO N
        B(J) - 1 ..
          DC K = I TO J - 1 ».

/* IF A(K) IS LESS THAN A(J) AND THE LENGTH OF THE LCNGEST   •/
/* MCNC TONE INCREASING SFCUENCE ENDING WITH A(K) EQUALS     */
/* CR IS GREATER THAN THE LCNGEST SEQUENCE CURRENTLY ENDING */
/* WITH A(J), THEN LENGTHEN IH€ SEQUENCE ENDING WITH A<J>    */

          IF A(K) LT A(J) THfcN
            IF e(J) LT 8(K) + I THEN
              BIJ) » B(K) • 1 t.
          ENCt.
        IF MAXINC LT B(J) THEN KAXINC = BIJ) ».
        END , .
      RETURN ( VAX INC ) ».
   ENO MCNSEC

DC WHILE ( 1 ) t.
 GFT LIST ( Nt ( Ad) DO I * 1 TO N ) )
 PUT LIST ( •THE SEQUENCES ( A(I) DU I - 1 TO N )  ,
 • HAS A LCNGtiST MCNGTONIC INCREASING SUBSEQUENCE UH LtN0TH• »
   MCNSEQ( A, N ) ) t•
 END
 END PAIN ».
```

Gaussian Elimination

Why I've included this problem:

Gaussian Elimination is a well known and important technique for solving systems of simultaneous linear equations. - every student of 15-100 should know it.

A Gaussian Elimination program in PLAGO requires that you know how to systematically operate on the rows and columns of an array. These techniques you should know.

The Problem:

Both the problem of solving sets of linear equations and the method of Gaussian Elimination are discussed in the book by Forsythe, et al. (pp. 333-349).

Read and understand that material before proceeding.

Write structured statements corresponding to the flowchart on page 349.

Compare your structured statements with the body of the procedure, GAUSS, whose text follows.

GAUSS does not perform the partial pivoting operations described in the flowchart on page 349. Change the program so that it does perform this kind of pivoting.

It has been suggested that elimination could be performed so that all coefficients both below and ABOVE the main diagonal are eliminated. This would mean that the entire "back solution" process could be removed. Rewrite part of the program to do this. Compare the number of arithmetic operations required by both methods. That's right, compare them. Which method is more efficient? Can you think of any other reason why one method is better than the other?

```
MAIN.. PROCEDURE OPTIONS( MAIN ) ,.
DECLARE ( A(25, 25), C(25), X(25), EPS, TEMP, MULP ) FLOAT,
          ( N, I, J, K, L, L1 )  FIXED ,.

/* INPUT EPS, N, A, AND C */

GET LIST ( EPS, N, (( A(I,J) DO J = 1 TO N ) ,
               C(I) DO I = 1 TO N ) ) ,.

PUT LIST ((( A(I,J) DO J = 1 TO N ),'/',
              C(I) DO I = 1 TO N ) ) ,.
```

```
ELIM..
  DO I = 1 TO N - 1 ,.
    DO J = I + 1 TO N ,.
    IF ABS( A(I,I) ) LE EPS THEN
      DO,.
        DO L = I + 1 BY 1 WHILE ( ABS( A(I,I) ) LE EPS & L LE N ) ,.
        IF ABS( A(L,I) ) GT EPS THEN
          DO,.
            DO L1 = I TO N ,.
            TEMP = A(I,L1) ,.
            A(I,L1) = A(L,L1) ,.
            A(L,L1) = TEMP ,.
            END ,.
          TEMP = C(I) ,.
          C(I) = C(L) ,.
          C(L) = TEMP ,.
          END ,.
        END ,.
      IF ABS( A(I,I) ) LE EPS THEN
        DO,.
        PUT SKIP LIST ( 'SINGULAR SYSTEM////') ,.
        STOP ,.
        END ,.
      END ,.
    MULP = A(J,I) / A(I,I) ,.
      DO K = I + 1 TO N ,.
      A(J,K) = A(J,K) - MULP * A(I,K) ,.
      END ,.
    C(J) = C(J) - C(I) * MULP ,.
    END ,.
  END ELIM ,.

IF ABS( A(N,N)) LE EPS THEN
    DO ,.
    PUT SKIP LIST ( 'SINGULAR SYSTEM////' ) ,.
    STOP ,.
    END ,.

 /* PERFORM THE BACKSOLVING PROCESS */

 BACKSOLV..
   DO I = N BY -1 TO 1 ,.
   X(I) = C(I) ,.
     DO J = N BY -1 TO I + 1 ,.
     X(I) = X(I) - X(J) * A(I,J) ,.
     END ,.
   X(I) = X(I) / A(I,I) ,.
   END BACKSOLV ,.

 PUT DATA (( X(I) DO I = 1 TO N )) ,.
 END ,.
```

Matrix Multiplication

Why I've included this problem:

Matrix multiplication is a useful thing to know.

Recent work in the area of computational complexity has revealed some new and more efficient algorithms for performing matrix multiplication. I think they are interesting. I also think they form the basis of some good programming exercises.

The Problem:

The product of matrix A, having M rows and N columns, and matrix B, having N rows and P columns, is a matrix, C, having M rows and P columns, where

$$C_{i,k} = \sum_{j=1}^{N} A_{i,j} * B_{j,k}$$

That's all!

The procedure called, DEFN, which follows, performs exactly this computation.

Unfortunately, as M, N, and P grow large, the number of computations grows "very" large. In particular, if M=N=P, the number of multiplications alone equals N cubed! Hence, enormous amounts of time can be spent multiplying even relatively small matrices.

Question: Are there better ways of multiplying matrices.

As it turns out, it wasn't until 1968 that any significant improvement was made over just the definition. At that time, S. Winograd presented a method which can multiply matrices with about half the number of multiplications used by the definition. He achieved this saving by noting that real multiplication is commutative and that some of the multiplications could be traded for additions. The method is based on the following identity:

$$\sum_{j=1}^{2\lfloor N/2 \rfloor} A_{i,j} B_{j,k} = \sum_{j=1}^{\lfloor N/2 \rfloor} (A_{i,2j-1} + B_{2j,k}) * (A_{i,2j} + B_{2j-1,k}) - \sum_{j=1}^{\lfloor N/2 \rfloor} A_{i,2j-1} * A_{i,2j} - \sum_{j=1}^{\lfloor N/2 \rfloor} B_{2j-1,k} * B_{2j,k}$$

where X means the greatest integer Y X.

If N is even then the left side is just the i,k-th element of C. Otherwise the product

$$A_{i,N} * B_{N,k}$$

must be added to thr expression.

Admittedly, the expressions look much more complicated than the original definition. The savings accrue by observing that the last two sums are dependent upon I and K respectively and need be computed just once at the beginning of the program. Thereafter, the number of multiplications is half that required by the definition.

Compute an "operation count" of exactly the number of additions and multiplications that would be required by both methods. These computations should be functions of M, N, and P.

The procedure called WINOGRAD multiplies two matrices using Winograd's method. Study it.

For what values of M, N, and P would you expect WINOGRAD to execute more rapidly than DEFN? Note that M, N, and P will be larger than you might expect. Why?

Can you imagine situations where the accuracy of the results from WINOGRAD would be poorer than those from DEFN?

* * *

In 1969, in a paper by Strassen, ("Gaussian Elimination is Not Optimal ", Numerische Mathematik 13, pp 354-356) a method was presented which could multiply two 2 x 2 matrices using just 7 multiplications instead of the usual 8, and which didn't require that multiplication be commutative. His identities look just awful. And here they are:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} * \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

then

$$C_{11} = Q_1 - Q_3 - Q_5 + Q_7$$
$$C_{12} = Q_4 - Q_1$$
$$C_{21} = Q_2 + Q_3$$
$$C_{22} = -Q_2 - Q_4 + Q_5 + Q_6$$

where

$$Q_1 = (A_{11} - A_{12})\,B_{22}$$
$$Q_2 = (A_{21} - A_{22})\,B_{11}$$
$$Q_3 = A_{22}\,(B_{11} + B_{21})$$
$$Q_4 \quad A_{11}\,(B_{12} + B_{22})$$
$$Q_5 = (A_{11} + A_{22})(B_{22} - B_{11})$$
$$Q_6 = (A_{11} + A_{21})(B_{11} + B_{12})$$
$$Q_7 = (A_{12} + A_{22})(B_{21} + B_{22})$$

Strassen provides no motivation or intuition as to how he ever found these. However, everywhere I've ever seen these things presented, the commentator has suggested a different mnemonic device to help reconstruct them. Find one for yourself! These identities can be used to multiply matrices of any size if they are used recursively on matrices whose elements are themselves matrices. Try writing such a program. You'll learn much.

```
/* MATRIX MULTIPLICATION BY THE STANDATE DEFINITION */

DEFN.. PROCEDURE ( A, B, C, M, N, P ) ,.

  DECLARE ( I, J, K, M, N, P ) FIXED ,.
  DECLARE ( A(*,*), B(*,*), C(*,*) ) FLOAT ,.
  DECLARE (T ) FLOAT ,.

NEST.. DO I = 1 TO M ,.
  DO K = 1 TO P ,.
    T = 0 ,.
    DO J = 1 TO N ,.
    T = T + A(I,J) * B(J,K) ,.
    END ,.
    C(I,K) = T ,.
  END NEST ,.
  END DEFN ,.

/* MATRIX MULTIPLICATION USING WINOGRAD'S METHOD */

WINOGRAD.. PROCEDURE (A, B, C, M, N, P ) ,.

  DECLARE ( M, N, P, I, J, K, N2 ) FIXED,
     BB FIXED ,
  ( A(*,*), B(*,*), C(*,*), AI(M), BK(P) ) FLOAT ,.
/* COMPUTE THE SUMS OF THE THINGS WE WANT TO THROW AWAY */

  N2 = 2 * FLOOR ( N / 2 ) ,.
  DO I = 1 TO M ,.
    T = 0 ,.
      DO J = 1 BY 2 TO N2 ,.
      T = T + A(I,J) * A(I, J + 1 ) ,.
      END ,.
    AI(I) = T ,.
    END ,.

    DO K = 1 TO P ,.
    T = 0 ,.
      DO J = 1 BY 2 TO N2 ,.
      T = T + B(J,K) * B(J + 1,K) ,.
      END ,.
    BK(K) = T ,.
    END ,.

    BB = ( N2.NE N ) ,.
WORK.. DO I = 1 TO M ,.
    DO K = 1 TO P ,.
    IF BB THEN T = A(I,N) * B(N,K) ,.
         ELSE T = 0 ,.
      DO J = 1 BY 2 TO N2 ,.
        JP1 = J + 1 ,.
        T = T + (A(I,J) + B(JP1,K))* (A(I,JP1) + B(J,K) ) ,.
        END ,.
    C(I,K) = T - AI(I) - BK(K) ,.
  END WORK,.

END WINOGRAD ,.
```

The Eight Queens Problem

Why I've included this problem:

This problem has been analyzed in a step-wise way which is instructive.

Its solution can be expressed recursively.

The Problem:

Dijkstra has devoted a chapter to the problem of the eight queens. Read and understand that chapter before you proceed with the text below.

Dijkstra chose to find all the ways of positioning eight queens on a chess board so that no queen was attacked by any other. The program below, again by Dijkstra, can be used to find just one solution to the problem. How can it be modified so that all possible solutions are found? Study the program carefully. Its data structures are the same as the program in EWD316.

Exercise

Suppose the problem is generalized to consider a rather stylized chess board consisting of N x N squares on which we wish to place N queens so that none is under attack. Modify the program to solve this problem. Are there any statements you can make about the existence or non-existence of solutions for arbitrary N?

```
TRYC.. PRCCEDURE ( J ) ,.
   DECLARE ( I, J ) FIXED ,.

   DC I = 1 TO 8 WHILE ( NOTSAFE ) ,.
     SAFE = A(I) & B( I + J ) & C( I - J ) ,.
     IF SAFE THEN
       GUTS..
         DO ,.
       A(I), B(I+J), C(I-J) = 0 ,.
         X(J) = I ,.
       IF J LT 8 THEN
         DO ,.
       SAFE = 0 ,.
         CALL TRYC( J + 1 ) ,.
         END ,.
       IF NOTSAFE THEN A(I), B(I + J), C(I - J) = 1 ,.
         END GUTS,.
       END ,.
     END TRYC ,.
```

The Towers of Hanoi

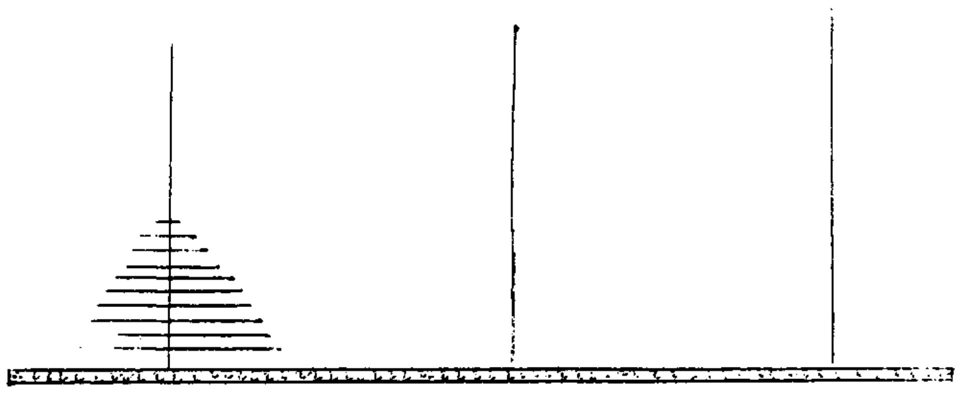
Why I've included this problem:

> This problem can be solved by a short, natural, recursive algorithm which you should understand.
>
> The problem has a nice generalization which I like.

The Problem:

Dijkstra devotes a section of EWD316 to this problem. His discussion, however, is somewhat more tedious than the one which follows. Read the text below, through the recursive solution to the problem. Then, read the section from EWD316. Finally, examine the program which solves the generalization to the problem.

Suppose that three spikes are driven into a flat board and that N doughnut-shaped discs have been arranged on one of the spikes with the smallest disc on top to the largest disc on the bottom. The diagram illustrates the situation.



The object of the game is to transfer all the discs from the starting spike to one of the other spikes so that they are left in the same order - smallest on top to largest on the bottom. The discs, however, may only be moved one at a time from one spike to another so long as a disc never rests on another disc of smaller diameter. That's the game!

The problem is to write a program that will produce a sequence of moves which will tell a player how to move each disc.

Clearly, if we have just one disc, the sequence of moves is trivial. Just move the one disc to one of the other spikes (designated as the finish spike).

If we have two discs, the situation is almost the same, except that the top disc must be moved to the intermediate spike; the bottom disc to the finish spike ; and finally the disc on the intermediate spike to the finish spike.

This suggests that to move N discs, we should:

(1) Move N - 1 discs from the start spike to the intermediate spike.

(2) Nove disc N from the start spike to the finish spike.

(3) Move N - 1 discs from the' intermediate spike to the finish spike.

The following program does exactly this.

```
 H C .  PROCEDURE  OPTIONS  (  MAIN  )  ».

HXNCI..  P R O C E D U R E " N , S ,  If  F  )  7»
   CECLARE  <N,  S,  I,  F)  FIXEO  , .
 7*  HANCT  COMPUTES" AND  PRINTS  A  SEQUENCE  OF  MOVES  WHICH  TRANSFERS    */
 /*  A  PILE  CF  <  N  )  DISCS  FROM  A  START  SPIKE,  S,  TO  A  FINISH  SPIKE,  */
 /*  F,  USING  SPIKE,  I,  AS  INTERMEDIATE  STORAGE.                          ""  */

   17  N~=⁻I  TFEN                                                      "
       PUT  SKIP  LIST  (  'MOVEJIISC      1,  •  FROM  «,  S,  •  TC  •,  F  )  ,.
       ELSE
         CO  , .
             CALL  HANOI  <  N  -  1,  S,  F,  I  )
         PUT  SKIP  LIST  t  *HCV£  DISC  ',  N,  '  FROM      S ^ i T ^ S J  )  ,.
             CALL  HANOI  (  N  -~T,  T V  S ,  F  )             "      "      ~     "
         END  , .                       _
     ENC  HANCI  ,.

    CECLARE  (  N  )  FIXED
_____CCWHJLE  (  1  =  1_1          _      ________________________________
   "GET  "LIST  ( ¥ 1
    CALL  HANCK  N,  1,2,3)  , .
    END  , . "
  ENC HC
```

Could the 'PUT LIST statement which specifically says to move disc one be eliminated?

What is the minimum number of moves necessary to move N discs? Find a formula which is a function of N and prove that it is correct.

Find a non-recursive algorithm which solves this problem. Which do you feel is the superior? Why?

```
MAIN.. PROCEDURE OPTIONS ( MAIN ) ,.
DECLARE ( I, N, NSPIKES, S, F, ISN ( 50 ) ) FIXED ,.

 GENHAN.. PROCEDURE ( N, NSPIKES, S, F ) ,.
    DECLARE ( I, N, NS, NSPIKES, S, F, FT ) FIXED ,.
    IF N LE NSPIKES - 1 THEN
    DO ,.
    DO I = 1 TO N - 1 ,.
    PUT LIST ( 'MOVE DISC ', I, ' FROM ', S, ' TO ', ISN(I)) ,.
    END ,.

    PUT LIST ( 'MOVE DISC ', N, ' FROM ', S, ' TO ', F ) ,.

    DO I = N - 1 BY -1 TO 1 ,.
    PUT LIST ( 'MOVE DISC ', I, ' FROM ', ISN(I), ' TO ', F ) ,.
    END ,.
    END ,.
    ELSE
  WORK.. DO ,.
    FT = ISN ( 1 ) ,.
    ISN(1) = F ,.
    CALL GENHAN( N-1, NSPIKES, S, FT) ,.
    PUT SKIP LIST ( ' MOVE DISC ', N , ' FROM ', S, ' TO ', F ) ,.
    ISN(1) = S ,.
    CALL GENHAN ( N-1, NSPIKES, FT, F ) ,.
    ISN(1) = FT ,.
    END WORK ,.
    END GENHAN ,.

  DO WHILE ( 1 ) ,.
    GET LIST ( N, NSPIKES, S, F ) ,.
    J = 1 ,.
    DO I = 1 TO NSPIKES ,.
      IF I NE S & I NE F THEN
        DO ,.
        ISN(J) = I ,.
        J = J + 1 ,.
        END ,.
      END ,.
    PUT SKIP LIST ( 'N= ', N, ' NSPIKES= ', NSPIKES, 'S= ', S,
    'F= ', F ) ,.
    CALL GENHAN( N, NSPIKES, S, F ) ,.
    END ,.
  END MAIN ,.
```

Suppose the problem is modified so that we allow a parameter which specifies the number of spikes the game will have. Thus the original game is a special case of of this more general one - in that game, the number of spikes was equal to 3.

What is the minimum number of moves necessary to move the N discs if you are allowed to use NSPIKES spikes?

A program follows which performs this computation. Can it be shortened? How?

What would a non-recursive algorithm look like?

The Coin Problem

Why I've included this problem:

> This problem has a very natural and intuitive recursive solution which can suggest a non-recursive solution which isn't quite so intuitive. I think you should see it.
>
> The problem also generalizes nicely.

The Problem:

Determine the number of distinct ways an arbitrary number of cents, A, can be "changed" in terms of half dollars, quarters, dimes, nickels and pennies. For example, 16 cents can be changed in exactly six ways, as:

(1) 16 pennies
(2) 11 pennies and 2 nickels
(3) 6 pennies and 2 nickels
(4) 1 penny and 3 nickels
(5) 6 pennies and 1 dime
(6) 1 penny and 1 nickel and 1 dime

How can the problem be analyzed? Consider first the notation:

$$N_A^C$$

which is interpreted as:

> "the number of ways of changing A cents with coins having maximum denomination C cents"

Thus the original problem is to find the value represented by the symbol

$$N_A^{50}$$

since we wish to change A cents with coins having maximum denomination 50 cents.

Now observe that

$$N_A^{50} = N_A^{25} + N_{A-50}^{25} + N_{A-2*50}^{25} + \cdots N_{A-\lfloor \frac{A}{50} \rfloor *50}^{25}$$

What does this mean? Just this: the number of ways of changing A cents equals the number of ways of changing A cents without any half dollars plus the number of ways of changing A cents using one half dollar plus the number of ways using two half dollars and so on.

Now note that each sub-problem on the right is similar to the problem with which we started except that there are fewer coin denominations to consider! Now notice these equations:

$$N_A^{25} = N_A^{10} + N_{A-25}^{10} + \cdots + N_{A-\lfloor \frac{A}{25} \rfloor * 25}^{10}$$

$$N_A^{10} = N_A^{5} + N_{A-10}^{5} + \cdots + N_{A-\lfloor \frac{A}{10} \rfloor * 10}^{5}$$

$$N_A^{5} = N_A^{1} + N_{A-5}^{1} + \cdots + N_{A-\lfloor \frac{A}{5} \rfloor * 5}^{1} .$$

What is the value of each term on the right of the last equation? Just 1. Surprise! In any case, the following prugram uses a recursive procedure to solve this problem based on the preceding analysis. Understand it.

```
CHANGE.. PROCEDURE OPTIONS ( MAIN ) ,.
        DECLARE COINS(5) FIXED ,.
  DECLARE A FIXED ,.
 WAYS.. PROCEDURE ( N, A) RETURNS ( FIXED ) ,.
        DECLARE ( N, A, TOTAL, I ) FIXED ,.
        IF N = 1 THEN RETURN ( 1 ) ,.
        ELSE
          DO ,.
          TOTAL = WAYS ( N - 1, A ) ,.
          DO I = 1 BY 1 TO FLOOR ( A   / COINS ( N ) ) ,.
            TOTAL = TOTAL + WAYS ( N - 1, A - I * COINS( N ) ) ,.
            END ,.
          RETURN ( TOTAL ) ,.
          END ,.
        END WAYS ,.

 COINS(1) = 1 ,. COINS(2) = 5 ,. COINS(3) = 10 ,. COINS(4) = 25 ,.
 COINS(5) = 50 ,.

  DO WHILE (1 = 1 ) ,.
          GET LIST ( A ) ,.
          PUT LIST ( ' THE AMOUNT ', A,
              ' CAN BE REPRESENTED IN ', WAYS( 5, A), ' WAYS' ) ,.
        END ,.
  END CHANGE ,.
```

Which program would execute more efficiently for large values of A? Which program would be easier to explain to someone who had never considered the problem?

Suppose that we wish to add a third parameter to WAYS, i.e.

WAYS(ND, N, A)

where ND elements of COINS will contain distinct coin denominations in ascending order, such that COINS(1) always equals 1. Thus the original problem would just be

WAYS (5, 5, A)

for some amount A. This generalization allows one dollar bills, five dollar bills, ten dollar bills, etc. to be considered in the computation of the number of ways of changing an amount A. Which of the above programs can easily be modified to handle this generalization? Write a non-recursive procedure which computes WAYS(ND, N, A). (Hint: use the elements of an array to behave like the controlled variables of a nest of DO-loops)

Suppose you wished to compute not only the number of distinct ways of changing an amount A, but also precisely what those ways are. What problems arise when you attempt to change the programs?

* * *

A PARTITION of a positive integer, A, is a sequence of positive integers whose sum is A. Use the ideas of the above programs to write both recursive and non-recursive programs which compute the partitions of A so that no computed partition is a permutation of some other partition.

Try simulating the behavior of the program for a few simple examples. Then observe that the following program also solves the problem.

```
CHANGE.. PROCEDURE OPTIONS ( MAIN ) ,.

  WAYS.. PROCEDURE ( A ) RETURNS (FIXED ) ,.
    DECLARE (A, I1, I2, I3, I4, TOTAL ) FIXED ,.
    TOTAL = 0 ,.
    NEST.. DO I1 = 0 BY 1 TO FLOOR ( A / 50 ) ,.
           DO I2 = 0 BY 1 TO FLOOR (( A - I1 * 50 ) / 25 ) ,.
           DO I3 = 0 BY 1 TO FLOOR (( A - I1 * 50 - I2 * 25 ) / 10 ) ,.
           DO I4 = 0 BY 1 TO FLOOR (
                    (A - I1 * 50 - I2 * 25 - I3 * 10 ) / 5 ) ,.
           TOTAL = TOTAL + 1 ,.
           END NEST ,.
    RETURN ( TOTAL ) ,.
    END WAYS ,.

  DECLARE ( A ) FIXED ,.
  DO WHILE (1 = 1 ) ,.
  GET LIST ( A ) ,.
  PUT LIST ( ' A = ', A, ' CAN BE CHANGED IN ', WAYS ( A ), ' WAYS' ) ,.
      ,.
  END ,.
  END CHANGE ,.
```

Counting Lattice Points

Why I've included this problem:

> This problem has a very natural recursive solution. I think you should see it.

The Problem:

We can define a lattice point in N-dimensional cartesian space as a set of N coordinates which are all integers. For example, in 2-space (just a plane) (-2, 0) is a lattice point but (.5, 2) is not a lattice point. The problem can now be stated.

> How many lattice points are contained in an N-dimensional hypersphere of radius R, centered at the origin.

That is, if N describes the dimension of the space and R describes the radius of the hypersphere, the algorithm should produce the number of lattice points within the hypersphere.

Consider first the cases which can be visualized. If N is 0, then there is exactly one lattice point, the origin, regardless of R.

If N is 1, then our space is just a line centered at 0, and the number of lattice points is just 2 * FLOOR(R) + 1. Another way of viewing the problem would be to count all the answers to the zero-dimensional problems which occur at the origin and to the right and left of the origin for integer I such that R**2 - I**2 is greater than or equal to 0. That is, count the origin just once and then count the points on either side, recognizing that this value is just twice the number to the right, say.

If N is 2, then our space is a plane, and the hypersphere is a circle of radius R, centered at the origin. Thus, the lattice points are all (u,v) such that u squared plus v squared is less than or equal to R squared and where u and v are both integers. Another view regards the problem in terms of a bunch of one-dimensional problems , i.e. count the number of lattice points on the x-axis and add to this twice the number of lattice points in the upper semi-circle.

The three dimensional case is thus just a bunch of two dimensional problems.

The program which follows performs the desired computations. Study it. Note that R**2 is passed as a parameter rather than just R. Why.

Find a non-recursive solution to this problem. Be careful!

Simulate the recursive structure of this program by maintaining your own stack.

```
LATTICE.. PROCEDURE OPTIONS ( MAIN ) ,.
   POINTS.. PROCEDURE ( N, RS) RETURNS ( FIXED ) ,.
/* POINTS COMPUTES THE NUMBER OF LATTICE POINTS IN AN N-DIMENSIONAL*/
/* HYPERSPHERE OF RADIUS SQRT( RS ) */
         DECLARE ( N ) FIXED, ( RS ) FLOAT ,.
         DECLARE ( S ) FIXED ,.
         IF N = 0 THEN RETURN ( 1 ) ,.
           ELSE
             DO ,.
             S = POINTS ( N-1, RS ) ,.
              DO I = 1 BY 1 WHILE ( I * I LE RS) ,.
                S = S + 2 * POINTS ( N-1, RS - I * I ) ,.
              END ,.
            END ,.
           RETURN ( S ) ,.
        END POINTS,.
   DECLARE ( N ) FIXED , (RS ) FLOAT ,.
   DO WHILE ( 1 = 1 ) ,.
   GET LIST ( N, RS ) ,.
   PUT LIST ( 'DIMENSION = ', N, ' RADIUS SQUARED = ', RS,
         ' NUMBER OF LATTICE POINTS = ', POINTS ((N), (RS))) ,.
   END ,.
   END LATTICE ,.
```